DOCUMENT RESUME

ED 100 370 IR 001 469

AUTHOR Claybrook, Billy G.
TITLE Learning as a Problem Solving Tool. Technical Report CS74018-R.
INSTITUTION Virginia Polytechnic Inst. and State Univ., Blacksburg. Dept. of Computer Science.
REPORT NO TR-CS74018-R
PUB DATE Nov 74
NOTE 44p.

EDRS PRICE MF-$0.75 HC-$1.85 PLUS POSTAGE
DESCRIPTORS *Artificial Intelligence; *Computer Programs; Cybernetics; *Learning Theories; *Problem Solving
IDENTIFIERS Claybrooks First Order Predicate Calculus; Heuristics; Watermans Production Rules; Winstons Representation

ABSTRACT

This paper explores the use of learning as a practical tool in problem solving. The idea that learning should and eventually will be a vital component of most Artificial Intelligence programs is pursued. Current techniques in learning systems are compared. A detailed discussion of the problems of representing, modifying, and creating heuristics is given. Some of the questions addressed in the paper are: (1) how does the choice of representation affect the potential for learning? (2) what techniques have been used to date and how do they compare? (3) exactly how are heuristics modified in the existing systems and what do these techniques have in common? A discussion of the credit assignment problem as it relates to learning under the various schemes of representation is also presented. (Author/DGC)

BEST COPY AVAILABLE

ED 100370

Computer Science Department

Virginia Polytechnic Institute
and State University
Blacksburg, Virginia 24061

IR 001 469
IR 001 469

Technical Report CS74018-R

LEARNING AS A PROBLEM SOLVING TOOL

Billy G. Claybrook

November 1974

U S DEPARTMENT OF HEALTH
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

BEST COPY AVAILABLE

LEARNING AS A PROBLEM SOLVING TOOL

Billy G. Claybrook
Computer Science Department
Virginia Polytechnic Institute & State University
Blacksburg, Virginia 24061

This paper explores the use of learning as a practical tool in problem solving. The idea that learning should and eventually will be a vital component of most Artificial Intelligence programs is pursued.

Current techniques in learning systems are compared. A detailed discussion of the problems of representing, modifying, and creating heuristics is given. Some of the questions asked (and answered) in the paper are: (1) how does the choice of representation affect the potential for learning?, (2) what techniques have been used to date and how do they compare?, i.e. first-order predicate calculus vs. production rules vs. Winston's representation, and (3) exactly how are heuristics modified in the existing systems and what do these techniques have in common? A discussion of the credit assignment problem as it relates to learning under the various schemes of representation is also presented.

1.0 INTRODUCTION

BEST COPY AVAILABLE

This paper is concerned with encouraging the use of learning as a problem solving tool in heuristic programs. We try to accomplish this: (1) by pointing out what has been accomplished, (2) by discussing what the major problems are, and (3) by showing how the problems can be approached.

The utility of heuristic programs depends to a large extent on the adequacy of the heuristics employed. We discuss three current techniques for representing heuristics that have been employed in successful, non-trivial program environments: (1) Waterman's production rules in playing draw poker[21], (2) Claybrook's first-order predicate calculus in factoring multivariate polynomials [2], and (3) Winston's representation in scene analysis [22].

2.0 PAST AND PRESENT ACCOMPLISHMENTS IN LEARNING

Except for a few learning programs, most learning programs to date, e.g. Michie and Ross' GT [9], Slagle and Farrell's MULTIPLE[18], etc., have been associated with simple problem domains. Notable exceptions are Samuel's checker program[17], Waterman's poker program[21], Claybrook's multivariate polynomial factorization program, and Winston's scene analysis program[22]. We feel that further advances in learning techniques could have been developed more readily had learning been implemented in more complex domains. We are not trying to reduce the importance of early research efforts, for they have provided us with a wealth of information.

Most learning programs have used some form of generalized learning [10] versus rote learning[10]. The early learning programs [9],[17],[18] implemented generalized learning by optimizing weights associated with problem variables

BEST COPY AVAILABLE

in linear evaluation functions. Samuel's program not only uses rote learning and generalized learning, but it also uses book learning as part of a training effort. Another type of learning that has received considerable use is concept learning [9], [20], [22]. Towster[20] provides several methods for programming concept-formation. Winston's program for learning structual descriptions from children's toys can recognize concepts and learn concepts to be recognized.

We have not tried to make an extensive survey of learning programs for they are well described elsewhere [4], [14], [19]. One fact that prevails throughout any study of learning efforts is that few of these learning techniques have been employed in solving practical problems for which algorithmic solutions do not exist or are very costly. Sammet [16] discusses several important problem areas in computer science where heuristic techniques could and should be employed.

It is interesting that Nilsson in his book[14] wanted to include a chapter discussing problem-solving methods using machine-learning techniques, but he concluded that the subject was not yet well enough developed to be included in a textbook.

3.0 MAJOR PROBLEMS WITH LEARNING

Since this paper encourages the use of learning in heuristic programs not only should we describe some of the major problems associated with learning, but also some of the reasons why people have failed to use learning.

There are several reasons why learning has not been utilized by problem solvers:

1. Not enough is known about learning by the human problem solver to use it.

2. The implementation of learning schemes, even simple ones, appears to be a formidable task; and in some instances it is.
3. There is always the possibility that the program will not learn or improve with experience, and thus not improve the efficiency of operation of the program.
4. The human problem solver may want to solve his problem as quickly as possible and is not interested in learning or its benefits.
5. There is a certain amount of overhead associated with learning, i.e. the process of analyzing solution attempts and modifying heuristics requires computer time.

Another purpose of this paper is to help remove some of the above problems.

The major problems associated directly with the actual use and implementation of learning techniques include:

1. Selection of a powerful representation of heuristics (requirements of a representation are given in Section 4.0).
2. Solution of the credit-assignment problem for modification and creation of heuristics.
3. Developing a training sequence appropriate for starting the learning process.
4. Determination of features of the problem environment on which to key the learning process.
5. Selection of the type of learning scheme to be used.
6. Evaluation of the learning effort (this is closely associated with the credit-assignment problem).

Some of the problems listed above need a closer look. The selection of a representation of heuristics and solution of the credit-assignment problem are discussed in detail in Section 4.0. The training sequence for a learning

program must be selected carefully for it influences the program's ability to learn and the rate of learning. Winston [22] argues the importance of a good training sequence and says that the sequence should be prepared by good teachers. His framework for learning suggests a unity between learning from examples, learning by imitation, and learning by being told. The training sequence can include samples for each type of learning. Winston also stresses the importance of the near miss. A near miss is a sample in a training sequence quite like the concept to be learned but which differs from that concept in only a few significant points at most. The near misses convey essential points much more directly that repetitive exposure to ordinary examples. Concept learning schemes normally require a careful selection of samples for a training sequence.

Claybrook [2] provides several examples from his multivariate polynomial factorization program, POLYFACT, that illustrate how the training sequence of polynomials can affect the factorization time of subsequently factored polynomials. Although the factorization times in POLYFACT are influenced by the training sequence of polynomials, the selection of samples is not as critical as it can be in other problem spaces and other learning programs. The reader should realize that the relative importance of a good training sequence is determined not only by the learning scheme used, but also by the representation of the heuristics and the characteristics of the problem environment.

Human beings tend to learn to solve a particular class of problems by keying on significant features of the problem class. The features are used to determine the approach taken in solving the problem. Unfortunately, there is no automatic way of extracting the significant features associated with a particular problem. The game of chess is a problem area where the features to key on for making a next move are extremely important. The human must almost always select the features to use in the learning process. The learning program can usually indicate which features are important for learning and "bad" features can

be removed from consideration. Sometimes the developers of learning programs believe that a large number of features will give better learning results; however, experience has shown [17], [18] that this is usually a misconception. Proper selection of a few "good" features will normally lead to good learning results. For example, only four features are used for term selection in POLYFACT and only seven or eight features are used for possibility selection. Waterman's state subvector for draw poker has only seven elements.

Selection of the type of learning, e.g. rote learning, generalized learning, or concept learning, is determined by characteristics of the problem environment. Rote learning has been used by Samuel with great success. His checker program stores records of board positions and when a move is to be made the previous board positions are interrogated to determine the proper move to make. An obvious disadvantage of rote learning is that large amounts of memory may be required to store the history. This is especially true if a large number of decisions are possible. As an example consider the number of possible board positions in chess or checkers. Another problem with rote learning is that time must be spent in retrieving and matching previous records with the current situation. Samuel has a sophiscated scheme for retrieving board positions.

A concept is a classification rule. Concept learning schemes also use features to classify objects. Concept learning schemes use past experience to classify an instance of an object as either positive or negative. This type of learning can also require excessive amounts of memory to store the past history. Concept learning techniques do not always satisfy the requirements for learning in a particular problem area. For example, Claybrook [2] used concept learning to try to determine the best possibilities (terms) to select during the creation of a factor in a polynomial. The results of this effort clearly demonstrated the shortcomings of concept learning in this particular situation. The reason for this is that "good" and "bad" possibilities have many features in common,

BEST COPY AVAILABLE

and it is difficult to classify one instance as positive and another as negative. What was required and eventually used was a generalized learning technique that was able to discriminate among the possibilities and rank them according to their apparent merit in creating a factor of a polynomial.

Most learning programs use generalized learning schemes (concept learning is actually a form of generalized learning) since they normally make use of previous experience without maintaining individual records of previous solution attempts. Waterman and Claybrook use generalized learning in their programs, but their modifications to heuristics involve much more than modifying weights in linear evaluation functions.

Since many learning programs [9], [17] are heuristic search programs, the methods for evaluating learning are associated with tree or graph measurements, e.g. measuring the bushiness (penetrance [8]) of the search tree or the path length from the start node to the goal node. These are gross measurements of learning. In a learning program such as POLYFACT, the learning in each area in the program, where learning occurs, should be evaluated individually with respect to the sense of direction that guides improvement through learning. It is the evaluation of learning that is used to determine modifications to heuristics.

4.0 PROBLEMS OF REPRESENTING, MODIFYING, AND CREATING HEURISTICS

This section addresses itself to the problems of modifying and creating heuristics using three representations: Waterman's production rules, Claybrook's first-order predicate calculus, and Winston's representation. First we discuss how the choice of representation affects the potential for learning. Then for each individual representation we describe: (1) the representation technique, (2) how the heuristics are created and modified, and (3) the credit-assignment

BEST COPY AVAILABLE

problem as it relates to learning under the representation. Finally, we compare the three techniques with respect to what they have in common and their power of representation.

For specific results associated with each of the three learning programs discussed in this section, we refer the reader to each author's dissertation in the list of references.

4.1 Choice of Representation

Since the representation of heuristics is probably the key to the success of any learning program, this prompts several considerations (or requirements) for selecting a representation:

1. The heuristics must be capable of representing complex actions in several problem areas.
2. The creation, modification, and execution of heuristics should be relatively simple tasks.
3. An appealing property of a representation scheme is that it conserve storage.
4. The representation should allow at least a partial solution of the credit-assignment problem.
5. The heuristics should be modular, i.e. the representation should allow the construction of heuristics from distinguishable components.
6. The representation should allow heuristics to be referenced as individuals or as members of designated sets of heuristics.
7. The heuristics should permit dynamic manipulation during program execution.
8. The final consideration is the flexibility of the representation, i.e.

the ability to interchange the components that comprise the heuristics in the event that heuristics are changed by the designer.

The reader may want to keep these considerations in mind while reading the rest of this section.

Some of the requirements require a brief explanation. The first consideration is motivated by the realization that many of the actions performed in complex learning programs such as POLYFACT require a comprehensive analysis of the problem situation, i.e. several criteria must be considered, often simultaneously, to assure that all prior conditions are satisfied before performing an action. Also, the first consideration suggests that we have a general representation that can be used in a learning system for solving various problems.

The second consideration does not necessarily imply that the decisions related to the actual creation and modification of heuristics be simple; however, once these decisions are reached for particular heuristics, the procedures should be mechanical in nature and relatively simple to execute. The third and fifth considerations are complementary. If the heuristics are modular, they can be represented in an encoded form to conserve storage. This is especially important in programs that use classification mechanisms for implementing localized learning. By localized learning, we mean that each classification has a set of learned heuristics for solving that particular class of problem.

Solution of the credit-assignment problem is included because of its importance to learning mechanisms. Representation of heuristics must enable the assigning of credit for success or failure among the many heuristics of potential use in solving a particular problem.

Dynamic manipulation, e.g. creating and modifying heuristics during program execution, is an absolute minimum requirement for a representation. The execution process performed on the heuristics must allow for dynamic changes in heuristics.

For this reason the execution procedures should execute the heuristics by using an interpretive process (LISP 1.5 and SNOBOL IV processors are interpreters and are therefore convenient languages for writing learning programs).

4.2 Waterman's Production-Rule Representation

Waterman's main interest is in devising machine-learning techniques that can be applied to the problem of learning heuristics. He sees the problem of implementing the machine-learning of heuristics as two subproblems:

1. Devise a method of representing heuristics that facilitates dynamic manipulation by the program using them.
2. Develop techniques by which a program can create, evaluate, and modify its own heuristics.

Waterman implemented his learning scheme, using a production rule representation for heuristics, in a draw poker playing program. He chose draw poker because it is a nontrivial game in which players do not have access to enough of the existing game information to perform effective minimaxing.

4.2.1 Representation of Heuristics

Waterman says that a good representation should:

(1) permit separation of the heuristics from the main body of the program,

(2) provide identification of individual heuristics and an indication of how they are interrelated, and

(3) be compatible with generalized schemes.

Two definitions are required for the following discussion:

Heuristic Rule. A heuristic that directly specifies an action to be taken.

Heuristic Definition. A heuristic that defines a term.

As a program is executed, it goes through a succession of states as the value of its program variables are changed. A state vector α is used to indicate the current values of the program variables. When a block of code is executed, the effect on the state vector α can be described by $\alpha' = f(\alpha)$, where α' is the resulting state vector and f represents a block of code. A heuristic is represented as a rule of the form $S \rightarrow T$ where S is the current state vector and T is the vector containing the mapped components. The rule can be thought of as a specification of how a state vector can lead to other state vectors. An example* is

$$\alpha = (A, 3, C) \rightarrow (g_1(\alpha), g_2(\alpha), g_3(\alpha)).$$

The function g, changes the value of A to $g_1(\alpha)$, B becomes $g_2(\alpha)$, and C becomes $g_3(\alpha)$. The items A, B, and C represent sets of values and not individual values. Thus, a single state vector such as (A, B, C) represents a number of states instead of a single state, thereby reducing the total number of heuristics required.

A rule of the type $S_b \rightarrow S_b'$, where S_b is a situation defined by vector variables and S_b' is the definition of the resulting situation. Production rules of this form are called action rules. A heuristic definition can be represented by a production rule of the type $Z \rightarrow Z'$, where Z is a value of a state vector variable and Z' is either:

1. a value of a state vector variable and an associated predicate (called a backward form rule), or
2. a computational rule for combining variables of the state vector (called a forward form rule).

An example of a backward form rule is $D1 \rightarrow D, D > 20$, meaning that D is considered a member of the set D1 if the current value of D is greater than 20. An example of a forward form rule is $X \rightarrow K1 * A$, meaning that X is defined by the arithmetic expression K1 * A.

* The examples in this section are taken from Waterman [21].

The state vector has three types of variables: (1) bookkeeping variables, which provide a record of past experiences; (2) function variables, which represent arithmetic expressions containg state vector variables; and (3) dynamic variables, which either directly influence the decisions of the program or change in value as a direct result of these decisions. Only dynamic variables are used in the descriptions which represent the left parts of action rules, but both dynamic and function variables are used in the right parts of these rules. The bookkeeping variables are used in the definitions of the forward form rules.

To gain insight into decision making through the use of Waterman's production rules we consider the following example from his paper. Let the subvector β be the following:

$$\beta = (A, B, C) = (a, b, c).$$

A, B, and C are dynamic variables with the current values of a, b, c, respectively. Consider the following simple heuristics:

1. If A is an A1, then add X to the value of B.
2. If A is an A2 and C is a C1, then subtract Y from the value of C.
3. If B is a B1, then add Y to the value of C.
4. A is an A1 when $A \geq 25$.
5. A is an A2 when $A < 25$.
6. B is a B1 when $B > 1$.
7. B is a B2 when $B > 4$.
8. C is a C1 when $C = 5$.
9. X increases as D increases.
10. Y increases as E decreases.

The corresponding production rules are:

1. $(A1, *, *) \rightarrow (*, X + b, *)$, action rule.
2. $(A2, *, C1) \rightarrow (*, *, c - Y)$, action rule.
3. $(*, B1, *) \rightarrow (*, *, Y + c)$, action rule.
4. $A1 \rightarrow A, \quad A \geq 25,$ backward form rule.
5. $A2 \rightarrow A, \quad A < 25,$ backward form rule.
6. $B1 \rightarrow B, \quad B > 1,$ backward form rule.
7. $B2 \rightarrow B, \quad B > 4,$ backward form rule.
8. $C1 \rightarrow C, \quad C = 5,$ backward form rule.
9. $X \rightarrow K1 * D,$ forward form rule.
10. $Y \rightarrow K2 - (K3 * E),$ forward form rule.

Also needed are the following rules, one for each element of the subvector:

11. $A \rightarrow a, \quad a \in \{\text{set of allowable values for A}\},$ backward form rule.
12. $B \rightarrow b, \quad b \in \{\text{set of allowable values for B}\},$ backward form rule.
13. $C \rightarrow c, \quad c \in \{\text{set of allowable values for C}\},$ backward form rule.

D and E are bookkeeping variables, and X and Y are function variables. A star (*) in the left side of an action rule indicates that the variable in question is irrelevant with regard to that particular situation description. A star (*) in the right hand side of an action rule indicates that the value of the variable in question remains unchanged. Thus, the action rule

$(A1, *, *) \rightarrow (*, X + b, *)$ means, if variable A has the symbolic value A1, then increment value of B by X.

Decision making by the program is done in two steps:

1. Each element of the current program subvector is matched against all right sides of the backward form (bf) rules. When a match occurs (the predicate is satisfied), the corresponding left side of that bf rule is matched against all right sides of bf rules, etc., until no more matches can be found. The resulting set of symbols defines a symbolic subvector.

2. The symbolic subvector derived in (1) is matched against all left sides of the action rules, going top to bottom, and when the first match is found the values of the program subvector are modified as described by the right side of the matched rule.

To illustrate decision making consider the following example:

Let the subvector have the values a = 4, b = 5, c = 6, the constants have values K1 = 1, K2 = 20, K3 = 3, and let the bookkeeping variables have the values D = 7 and E = 8. Then $\mathfrak{B}$ = (4,5,6).

Step 1 is started by comparing a = 4 with each bf rule predicate, the predicate being satisfied only if it contains the symbol a and is true when a is set equal to 4. Thus a = 4 is found to match rule 11 and no others. Next A = 4 is compared to the right hand side of all bf rule predicates and is found to match only rule 5. Finally A2 = 4 is compared with all bf predicates and since it matches none of them the search terminates leaving A2 as the final symbolic value. Elements b and c are processed in the same manner and the symbolic subvector that results is ((A2), (B1, B2), (C)). This subvector is a description of all situations in which variable A has the symbolic value A2, the variable B has either the symbolic value B1 or B2, and the variable C has the symbolic value C.

Step 2 now consists in comparing the symbolic subvector ((A2), (B1, B2), (C)) with the left side of each action rule until a match is found. In this case a match occurs at rule 3. The program subvector is then set to the values specified in the right side of rule 3. Hence the new B equals (4,5, (20 - (3 * 8)) + 6) or (4,5,2). The program makes one external decision for each search cycle. Thus in a game-playing task the program would execute one search cycle each time it made a "move".

The subvector Waterman used for the game of draw poker is composed of the dynamic variables of the state vector and has the form:

B = (VDHAND, POT, LASTBET, BLUFFO, POTBET, ORP, OSTYLE),

where VDHAND is the value of the program's hand, POT is the amount of money in the pot, LASTBET is the amount of money last bet, BLUFFO is a measure of the probability that the opponent can be bluffed, POTBET is the ratio of the money in the pot to the amount last bet, ORP is the number of cards replaced by the opponent, and OSTYLE is a measure of conservative style by the opponent.

4.2.2 Program Creation and Modification of Heuristics

To create heuristics (either by modifying existing ones or hypothesizing new ones), Waterman uses three pieces of information:

1. a good decision for the situation,
2. the subvector variables relevant to making this decision, and
3. the reason the decision is being made.

This data is called the training information. Item (1) above is called the acceptability information, item (2) is called the relevancy information, and item (3) is called the justification information. The training information is either supplied by a trainer or obtained by the program during execution. The training information provides data for the construction of a new action rule. The acceptability information supplies the right part of the action rule, and the relevancy and justification information supplies the left part.

When the existing action rules lead to a poor decision, they are corrected by incorporating the training information into the production rules. Either an existing action rule (target rule) is modified to catch the symbolic subvector, or if a rule appropriate for modification does not exist, the training rule is inserted in the action rule list immediately above the error causing rule.

BEST COPY AVAILABLE

An action rule is appropriate for modification if it has the same form as the training rule. Two action rules have the same form only if (1) their right parts are identical, (2) their left parts have corresponding *'s, and (3) their left parts have symbolic values which correspond to the degree that they are both defined by the same logical operator.

An example (taken from Waterman [20]) illustrates how an action rule can be modified to catch the symbolic subvector. The training information is: (1) a good decision to add 2 to the value of B, (2) the variables relevant to this decision are A and C, (3) the decision is being made because the current value of A is small and the current value of C is large. The program subvector is (5, 3, 13). The training rule (action rule) created from the training information, and the associated backward form rule are:

$$(\ A1,\ *,\ C1\ ) \rightarrow (\ *,\ b + 2,\ *\ ),$$
$$A1 \rightarrow A,\quad A < 6$$
$$C1 \rightarrow C,\quad C > 12$$

The existing production rules are:

1. $(A1,\ *,\ C2) \rightarrow (*,\ b + 2,\ *)$.
2. $(A1,\ B1,\ *) \rightarrow (*,\ *,\ a + 5)$.
3. $(A2,\ *,\ C3) \rightarrow (*,\ b + 2,\ *)$.
4. $(A1,\ *,\ *) \rightarrow (*,\ *,\ a + 5)$.
5. $A1 \rightarrow A,\ A < 6$.
6. $A2 \rightarrow A,\ A < 8$.
7. $B1 \rightarrow B,\ B > 8$.
8. $C1 \rightarrow C,\ C > 12$.
9. $C2 \rightarrow C,\ C < 5$.
10. $C3 \rightarrow C,\ C > 13$.

Rule 3 is the only action rule which has the same form as the training rule $(A1,\ *,\ C1) \rightarrow (*,\ b + 2,\ *)$. The symbolic subvector obtained through parsing is

((A1, A2), (B), (C1)), which catches on rule 4. Rule 4 leads to an unacceptable decision (the only acceptable decisions are those supplied by the training information). Rule 3 leads to an acceptable decision and has the same form as the training rule; thus, it is used as the target rule. The left hand side of the training rule, (A1, *, C1), matches the left side of rule 3 except for C3. If C3 in rule 3 is replaced by a symbolic value representing a set large enough to include the current value of state vector variable C, the symbolic subvector obtained through parsing will catch on rule 3. Therefore C3 is replaced by C1, changing rule 3 to (A2, *, C1) $\longrightarrow$ (*, b + 2, *).

Waterman gives the following training procedure outline:

1. Parse the program subvector to obtain the symbolic subvector. Then drop this symbolic subvector through the action rules to obtain a decision. If the trainer indicates that the decision is acceptable, then stop; otherwise, go to step 2.
2. Obtain the training information from the trainer and use it to construct the training rule. If this information changes the symbolic subvector, then go to step 3; otherwise, go to step 4.
3. Drop the new symbolic subvector through the action rules to obtain a decision. If the decision is the one sought by the acceptability information, then stop; otherwise, go to step 4.
4. Locate the error-causing rule, the action rule responsible for the unacceptable decision made in step 1 or step 3.
5. Search the action rules above the error-causing rule for a target rule, a rule which has the same form as the training rule and is suitable for modification to catch the symbolic subvector. If such a rule is found, modify it to catch the symbolic subvector and go to step 3; otherwise go to step 6.

6. Search the action rules below the error-causing rule for a target rule. If (1) such a rule is found, (2) the error-causing rule is suitable for modification to pass the symbolic subvector, and (3) the rules between the error-causing rule and the target rule either pass the symbolic subvector or are suitable for modification to pass it, then modify the target rule to catch the subvector, the error-causing rule to pass the subvector, and the rules between these two to pass the subvector, and go to step 3; otherwise go to step 7.
7. Place the training rule immediately above the error-causing rule in the list of action rules and stop.

The learning in Waterman's program is in two forms: (1) learning with explicit training, or (2) learning without explicit training (implicit training). When the program learns without explicit training, the program itself must develop the training information during the course of game play. The acceptability information for implicit training can be obtained through logical deduction. This process uses:

1. the rules of the game,
2. statements (or axioms) about the game, and
3. general statements about techniques used in game playing.

The result is a set of logical statements from which new statements can be deduced using deductive inference rules. The reader is referred to Waterman's paper for an example of an actual deduction.

The justification information for implicit training can be obtained from a decision matrix that is game-dependent and is given to the program before learning starts. Each row of the matrix stands for a game decision, and each column stands for a subvector variable. Each entry E_{ij} in the matrix is an expression whose value is an attribute of the subvector variable j.

The relevancy information for implicit training is obtained through the generation and testing of hypothesis concerning the relevance of subvector

variables. Reasonable hypotheses are solved for in the following way:

1. Let the initial hypothesis for each rule be that all subvector variables are relevant.
2. Hypothesis testing then consists in noting whether or not a particular training rule, placed in the set of action rules by step 7 of the training procedure, catches the symbolic subvector when the action advocated by the rule is determined to be the correct decision.
3. If the rule does not catch the subvector, the relevancy hypothesis for that rule is changed. As many variables in the left part of the rule are made irrelevant as is necessary to make the rule general enough to catch the subvector.

4.2.3 Program Evaluation of Heuristics

Program manipulation of heuristics requires facing two major problems:

1. evaluation of existing heuristics in terms of their usefulness to the program, and
2. creation of new heuristics, by both modifying old ones and hypothesizing new ones.

To make a decision via production rules for a problem (1), a symbolic subvector representing the game situation is compared to all left parts of the list of action rules, going top to bottom until a match is found. The action rule which defines the decision, that is, one whose left part matches the symbolic subvector, is easily located. After the decision is evaluated, the credit or blame can be assigned to the action rule, and to those above it, which defined the decision. Here blame is assigned to action rules leading to

poor decisions, while action rules leading to good or acceptable decisions are ignored. Assigning blame to an action rule consists in modifying the rule enough to avoid a repetition of the mistake or poor decision just made.

4.3 Claybrook's First-Order Predicate-Calculus Representation

Complete details of this representation can be found in Claybrook [2], [3]. This representation is implemented in a learning program that performs the non-trivial task of determining the symbolic factorization of multivariate polynomials with integral coefficients and an arbitrary number of variables and terms. The author agrees with Waterman that the representation of heuristics determines directly or indirectly how well a program can learn. The representation in the learning program, POLYFACT, was chosen because of the expressive power of the predicate calculus. We were primarily concerned with using learning to improve the efficiency of operation of POLYFACT.

4.3.1 Representation of Heuristics

The notation is identical to that of first-order predicate calculus except for a minor difference involving domain specification for the assignment of values. In the implementation of the predicate calculus notation, a heuristic can have one of two general forms:

(1) NAME $(DOMAIN_1)$ $(DOMAIN_2)\cdots(DOMAIN_k)$ ($(ANTECEDENT_1$ C $CONSEQUENT_1)$ $0\cdots0$ $(ANTECEDENT_n$ C $CONSEQUENT_n))$ \$, or

(2) NAME $((ANTECEDENT_1$ C $CONSEQUENT_1)$ 0 $(ANTECEDENT_2$ C $CONSEQUENT_2)$ $0\cdots0$ $(ANTECEDENT_n$ C $CONSEQUENT_n))$\$

In either of the above forms, the same antecedent or consequent can occur several times; but the same antecedent-consequent pair should occur but once.

The first form has a non-null domain; whereas, the second has a null domain. One of the functions of the non-null domain is to specify an ordered set from which the values for the variable (indicated in the domain field) are taken. Some of the variables in the antecedent-consequent pairs can be free, i.e. their values are specified elsewhere. Each bound variable must appear as an argument in at least one antecedent or consequent, i.e. each variable specified in a domain must appear as an argument in at least one of a predicate, a function, or a consequent. The domain as defined in this paper corresponds to the quantifiers in predicate calculus notation; however, in predicate calculus notation the domain is not included as a part of the quantifier. The order of domain precedence is identical to that of the quantifiers.

Each antecedent is a single predicate or a logical combination of predicates connected by conjunction ('A' = AND) and/or disjunction ('O' = OR) operators. Each predicate is a logical function and can be referenced with arguments that are constants, variables, or functions. 'C' is the conditional operator, and the consequent is always the name of a routine (or procedure) that is executed when the corresponding antecedent is satisfied.

To illustrate the representation of heuristics in the predicate calculus notation, we use an example taken from the term selection heuristics in POLYFACT:

H1.1 (E T IN IPTRSO) ((N H1(G11(T), MINDEG) C FIX123)) $.

This heuristic consists of the components:

H1.1	is the NAME of the heuristic,
(E T IN IPTRSO)	is the DOMAIN of the heuristic,
N	is the negation operator,
H1	is a predicate that is 'TRUE' if G11(T) equals MINDEG,
G11	is a function whose value is the degree of term T,
T	is a bound variable

MINDEG	is a constant function, i.e. a function whose value is constant during the execution of the heuristic,
C	is the conditional operator, and
FIX123	is a CONSEQUENT.

Internally, the predicate calculus heuristics are represented as linked lists with each individual atom stored in a separate cell in the list.

The heuristics are executed by an interpreter. During execution, the predicate calculus form is translated into reverse Polish notation. Then the reverse Polish string is executed with references to predicates and consequences causing the execution of the corresponding procedures.

The heuristics that contain non-null domains select elements from the sets given in the domains. In the selection of elements from a set, a heuristic can consider all elements in the set. In this case, the domains have the form:

(E T IN IPTRSO),

where the E indicates that all elements (T) in the set IPTRSO are selected during the execution of the particular heuristic. A heuristic with a non-null domain can also consider elements from a set until an antecedent is satisfied. The corresponding consequent is then executed and activation of this heuristic is terminated. This type of domain is represented as:

(EA T IN IPTRSO),

where EA indicates that some (possibly all) elements in the set IPTRSO are selected.

A heuristic with multiple domains is executed by selecting elements from the innermost domains first. This execution has the same effect as nested loops in programming languages.

4.3.2 Program Modification and Creation

First we explain how the creation and modification process works, and then we describe the training procedure for POLYFACT. The reader saw in Section 4.2.2 that during periods of implicit training in Waterman's program, his program employs a decision matrix (created by a human prior to implicit learning). The learning scheme in POLYFACT uses a set of tables to specify relationships between predicates, consequents, and domains. The tables are pre-compiled by hand and read from input cards and stored in the tables.

The consequent-predicate table gives the correspondence between each consequent and the predicates that can be used to form an antecedent-consequent pair. The consequent-domain type table specifies the correspondence between each consequent and the sets from which values for a variable are selected. Each bound variable must be an argument in a predicate (within an antecedent) or consequent. The domain type-variable-set table defines the variable-set pair associated with a domain type. The domain is determined by the variable and the set from which the values of the variable are taken. The purpose of this table is to prevent heuristics with a given type of domain from using predicates and consequents associated with another type of domain. In addition this table could prevent the creation of heuristics which have a certain mixture of domains.

The reader should note that a domain is a set of values (represented in POLYFACT as a linked list of values), a predicate is a logical function (represented in POLYFACT as a logical procedure), and a consequent is an action to be taken (represented in POLYFACT as a procedure).

The heuristics in POLYFACT can be maintained either in first-order predicate calculus notation or in a combination of first-order predicate calculus notation and an encoding of the predicate calculus notation. As we describe the learning

associated with term and possibility selection we will describe how the predicate calculus is encoded.

The learning (through the modification of heuristics) associated with term selection is as follows. After a successful factorization attempt is completed, the number of possibilities (factors of a term) in each term of the polynomial is determined. The features of the term(s) with minimum number of possibilities have their frequency count(s) increased. Each feature has a predicate associated with it. The predicate is true if the term has the particular feature and false if it does not have the feature (features used are degree of term, number of variables in term, etc.). The frequency counts associated with each feature are examined to determine whether or not the set of heuristics for term selection need to be modified. The heuristics are ordered to impart the importance of features for "good" term selection. If two or more features have identical frequency counts, then they are of equal importance in selecting a term. Thus, in predicate calculus notation this would result in an antecedent having two predicates (corresponding to the two features) connected by'OR'.

Since POLYFACT uses a classification technique to implement localized learning for term selection,the term selection heuristics are not maintained in predicate calculus notation (because of the amount of storage space required). Instead, the term selection heuristic are encoded into a small ordered list of words. Each feature is represented by a particular bit in the word. The presence of a '1' in that bit indicates the presence of the corresponding predicate in the heuristic. In this way a single word describes the entire heuristic. Not only does this encoding save considerable storage space, but it is much easier to modify heuristics using a numeric representation than the symbolic representation of predicate calculus. Prior to execution the selected heuristics are expanded into predicate calculus notation for interpretation.

The possibilities (terms) that can be selected as terms in a factor of a polynomial are ranked according to their probable merit. During a factorization attempt, the highest ranked possibilities are selected. After a polynomial has been factored, each term in the factors is examined to determine its set of characteristic features. A binary vector is created with nonzero entries indicating the features present. Then a heuristic is created (unless one already exists) using as predicates those that correspond to the features present.

To facilitate the construction of this set of heuristics, a matrix is maintained providing a history of the features of terms that have appeared in factors of previous polynomials. After the vector of features has been created for a term, it is compared with each row in the matrix to determine if the vector is already present. If so, the frequency count for the matching row is incremented (the frequency count is kept as an augmented column in the matrix). If the vector is not in the matrix, it is added and the corresponding heuristic is created.

When these heuristics are used to rank terms, the satisfied antecedent's (if there is a satisfied antecedent) frequency count becomes the rank of the term. These heuristics are maintained in predicate calculus notation and also in the encoded matrix form. The matrix form is convenient for determining the need for modifications. All classes of polynomials use the same possibility selection heuristics. Modifications range from adding a predicate to an existing antecedent to adding an entire antecedent-consequent pair.

Heuristics can be created and modified during the training period or later when no explicit information is given POLYFACT. During the training period, polynomials are input to POLYFACT along with information giving the number of terms in each of the two factors. In this training period the polynomials are classified and heuristics are created for term selection and possibility selection. Polynomials in the training sequence do influence the factorization times of

subsequently factored polynomials, but similiar looking polynomials have so varied characteristics that it is difficult to select a good training sequence. However, after the training period is over, if the polynomials have many characteristics in commom then the program adjusts the heuristics to reflect this. During the non-training period no helpful information is given to POLYFACT. Also learning can be turned off completely at any time.

4.3.3 Program Evaluation of Heuristics

Assigning credit or blame to a heuristic in POLYFACT is a much simpler task than in Waterman's program, and the capability to reference heuristics individually by name provides the ability to do this.

Credit is given a heuristic by increasing the frequency count associated with the heuristic. Blame is not so easy to interpret. Blame can be interpreted when the term selection heuristics do not select the "best" term to initiate the factorization process, and when the possibility selection heuristics do not rank the possibilities so that only the highest ranked ones appear in the factors of a successfully factored polynomial.

Evaluation of heuristics can result in re-ordering heuristics, adding predicates to antecedents, etc. Term selection heuristics can be modified on either successful or unsuccessful factorization attempts, but possibility selection heuristics can only be altered after successful attempts.

4.4 Winston's Representation

Before we discuss Winston's learning system (or more properly his language and notation for describing scenes), we describe, in general terms, what his system does. Winston's program analyzes scenes consisting of the simple objects

found in a child's toy box. The description of a scene is in terms of the objects that make up the scene.

Generation of a scene description begins with a drawing of the three-dimensional scene. The drawing is communicated to the machine using a program together with a special pen whose position on a tablet can be read by the machine directly. Then a program classifies and labels the vertexes. The program then creates names for all of the regions in the scene.

Descriptions of scenes are stored so that they can be easily retrieved. Each object in a scene is naturally thought of in terms of relationships to other objects and to descriptive concepts like small, square, etc. Thus, Winston uses networks to store the scene description. The network resides in the data base in the form of list structures. For example, in Fig. 2 the nodes represent objects and the pointers represent relations between objects.

The descriptions permit one to compare and contrast scenes through programs that compare and contrast descriptions by retrieving the descriptions from the network. The descriptions should be similar or dissimilar to the same degree that the scenes they represent seem similar to dissimilar to human intuition. After two scenes are described and corresponding parts related by a matching program, differences in the descriptions must be found, categorized, and themselves described. Later sections describe how the matching of scenes is used to modify the models of scenes.

Identification of scenes is carried out as follows: compare the description of some scene to be identified with a repertoire of models or stored concepts. There is a method of evaluating the comparisons between the unknown and the models so that some match can be defined. The identification process in Winston's program is a major problem area. It is comparable to determining whether or not two graphs are isomorphic.

BEST COPY AVAILABLE

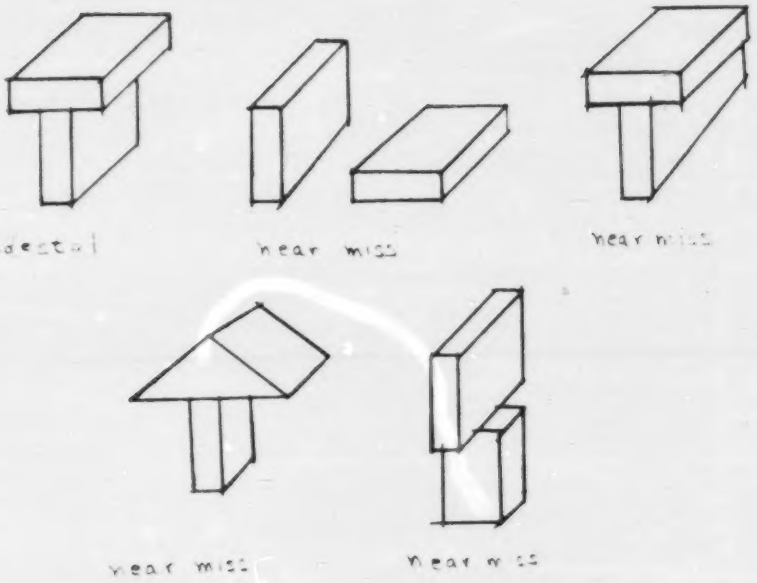

Fig. 1. A pedestal training sequence.

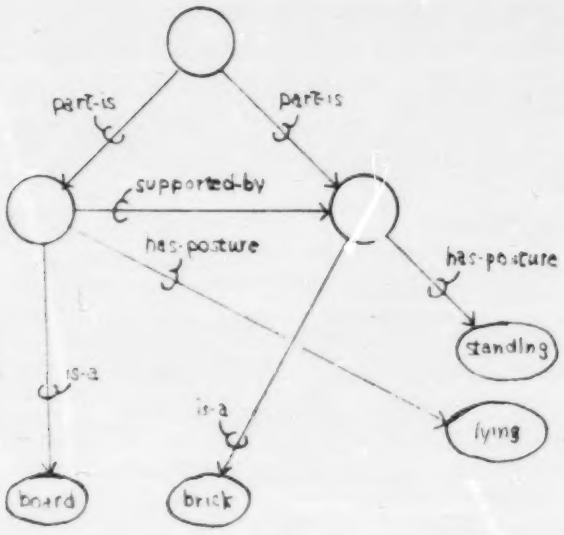

Fig. 2. A pedestal description.

BEST COPY AVAILABLE

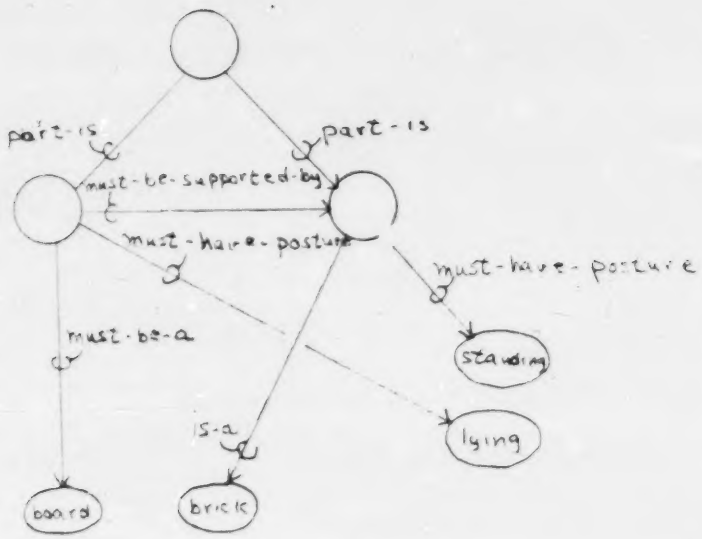

Fig. 3 A pedestal model

The next two sections provide more details for the construction of models (or concepts) and the language used to describe the models.

4.4.1 Representation of Models

It is difficult to talk about Winston's learning system with respect to his representation of heuristics. He stores learned information in a network (model) described by a language expressing relations between objects in a scene. The model represents (or is) the concept. During the building of a model of a scene such as that in Fig. 3, Winston's program is creating a concept to classify a scene.

There is a slight difference between a description of a particular scene and a model of a concept. A model is like an ordinary description in that it carries information about various parts of a configuration. But a model also exhibits and indicates those relations and properties that must and must not be in evidence in any example of the concept involved.

In order to develop the representation of models, we use the pedestal training sequence in Fig. 1. Then we describe a pedestal description and a model in Fig. 2 and Fig. 3, respectively (these examples are taken from Winston[23]). The first step is to show the machine a sample of the concept to be learned. The rest of the samples are near misses (a near miss is a sample in a training sequence like the concept to be learned but differs from that concept in only a few significant points). The near misses simply refine the description of the pedestal to the point where it is a model of the pedestal.

The second sample in Fig. 1 is a near miss due to the absence of the supported-by relation (a description of how the relations in a scene are determined is given in the next section). The other samples strength the other relations in the description and finally turn the pedestal description in Fig. 2 into the model in Fig. 3. The training sequence in Fig. 1 is revisited in the

next section on creation and modification of models.

This section was intended to give a brief view of the representation of a model. We mentioned earlier that the model is represented internally as a network using list structures.

4.4.2 Creation and Modification of Models

We have already discussed the difference between a scene description and a model. In this section we describe in detail how a model is created and modified during a training sequence. We use the pedestal training sequence in Fig. 1. Before we become specific on the development of the model in Fig. 3, perhaps we should consider a more general description of model development.

The model building program starts with a description of some example of the concept to be learned. This description is the first model of the concept. Fig. 4 illustrates the development of a model sequence where there is only one difference between the current model and the description of a new sample. Each new sample leads to a new model. Winston's program compares the description of the sample to the current model to determine any difference(s). We did

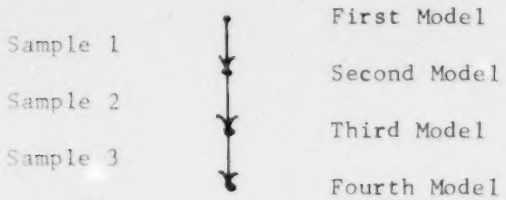

Fig. 4 Model Development with Only One Difference.

not point this out in Section 4.4, but during the development of the description of a scene, each scene is analyzed to determine the relations that objects in the scene have. A separate program exists for detecting each relation, i.e. a heuristic program exists specifically for detecting the existence of the SUPPORTED-BY relation, the IN-FRONT-OF relation, etc.

Several differences may occur between the current model and a new sample. Then several branches may occur and we have a tree of nodes as given in Fig. 5. The alternative branches come about by the program selecting one branch at each

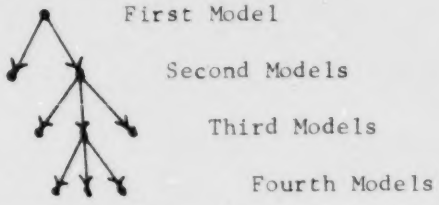

Fig. 5 Model Development with Several Differences,

point for further development. The path leading from the top of the tree down to the current model is called the main line. The main line changes course when a particular sequence of branch selections leads to untenable situations.

The program has to deal with alternatives to the main line of model development. Main line assumptions may lead to contradictions which in turn cause the model building program to retreat up the tree and attempt model development along other branches.

If the differences have multiple interpretations or more than two differences occur, the number of possibilities can explode. The machine must decide which interpretation of which differences are most likely to cause the near miss. The machine first forms two lists: a primary list and a secondary list. Each interpretation eventually ends up in one list or the other. Some interpretations can never make the primary list because they are unable to explain why a given sample is a near miss. All of these interpretations go immediately to the secondary list.

The next way to sort differences is by level. This assumes only that the differences nearer the origin of the comparision description are the more important. The program determines the depth of the remaining nodes which are nearest the origin of the comparison description. All those candidates found at greater depth are placed on the secondary list.

The primary difference list allows the program to form a theory of why the near miss misses and what to do. This theory (or hypothesis) specifies one difference as the single cause of the miss and specifies which interpretation of that difference is assumed. The differences at the same level are ranked according to type. Then the one with the highest rank is chosen as the cause of the near miss. Winston [24] provides a table specifying, a priori, differences and their possible interpretations.

Now we return to our discussion of the pedestal training sequence and model in Fig. 1 and Fig. 3. Reviewing very briefly, the model building program begins with a description of the concept to be learned (in Fig. 1). The second sample in Fig. 1 is a near miss because the supported-by relation is missing. Thus, the machine can only conclude that the supported-by relation is necessary and a new model is developed with a must-be-supported-by relation. We see there is only a single difference between the second sample and the current model. Samples three through five strengthen the fact that the support is standing and the supported object is a lying board. In particular, sample four strengths the relation that the supported object is a board and sample five strengthens the fact that the board must be lying. In samples four and five there is only a single difference.

The strengthening of the relations in Fig. 2 by the training sequence in Fig. 1 results in the pedestal model in Fig. 3.

The reader should be able to detect a note of importance to the development of a training sequence for model building. Winston stresses the importance of a good teacher both in human learning and machine learning. He says that in the past history of machine learning the use of a teacher was considered cheating, and machines were expected to self organize themselves. Winston's training sequence sample selection is probably more critical than the training sequence selection in Waterman's program, and it certainly is more critical than in Claybrook's POLYFACT.

The subsection on program evaluation of heuristics for Winston's system is omitted since this material is duscussed above in the development of models. This omission brings up an interesting point - some learning systems can be subdivided into very clear cut subdivisions, while others cannot.

4.5 Common Characteristics of the Three Techniques for Representing Heuristics

One characteristic common to all three techniques is that a powerful language is used for each representation. This is an especially important point because the early learning programs lacked powerful languages for heuristic representation, and it is the author's contention that this is the reason for the lack of significant advances in learning during the 1960's. Another point, historical in nature, is that all three techniques were developed in the early 1970's (i.e. all three dissertations were completed during this period).

Another common characteristic is that all three techniques were employed in complex problem spaces versus simple game problem spaces.

Each representation language is separated from the program code, i.e. the heuristics in each program are separated from the program code, thus allowing them to be manipulated dynamically. Also the representations are modular in nature allowing the heuristics to be easily created and manipulated. Although the approach in each case differs, each technique uses a form of generalized learning. Credit assignment occurs in each technique and is discussed in the next section.

All three techniques for representing heuristics have most or all of the requirements of heuristics listed in Section 4.1. The reader may want to scan this list again and consider each representation as he does so.

Another thing common to all three techniques (and also common to all other previously implemented learning systems) is that a change in problem environment

requires some effort on the part of the user to determine possible heuristics for solving the new problems (or at least determine features for keying on during learning) and in some cases providing the learning system, a priori, with rules or other information about the problem area. In Waterman's poker program, rules of the game are supplied and also a precompiled decision matrix is supplied, Claybrook supplies tables for controlling the construction of heuristics, and Winston provides tables listing one or more interpretations for each difference in scene descriptions. At the present time, it is nearly, if not completely, impossible to develop a learning system that can operate on various problem areas without some information being supplied to the system by the user. In Section 5.0 we discuss how to organize a learning system so as to reduce the effort in moving from one problem domain to another.

The last comparison of the three representation techniques is with respect to their power of representation. The predicate calculus and production rule representation languages appear to be the more powerful languages for representing complex actions. The network model approach is a natural approach to representation but it requires efficient retrieval and matching procedures to be practical. Winston acknowledges these two problems in his dissertation and has no "good" solution to them. Barrow [1] has made progress in structure matching.

The production rule system of Waterman appears to be the most complete system for machine learning of heuristics. Claybrook's system needs more development in the area of automatic generation of heuristics. Waterman's system also has the advantage that it is probably better documented in the literature than the other two systems.

4.6 Credit Assignment

The author feels that credit assignment in Waterman's representation is the most sophiscated and advanced; however, the reader should remember that this was

one of the main thrusts of his research, while Claybrook and Winston were more interested in studying a particular problem area.

It is not easy to discuss credit assignment with respect to Winston's language so we will begin by discussing credit assignment as it relates to learning under Claybrook's representation, followed by Waterman's representation, and finally Winston's representation.

The program POLYFACT has an analysis procedure that analyzes each factorization attempt and collects information on which features of polynomials appear to be the most important with respect to the chosen heuristic that directs the learning. One thing that every learning program must have to handle the credit assignment problem is some sense of direction for directing learning. Either the user provides this sense of direction, as in POLYFACT, or the program can possibly learn it. In POLYFACT assigning credit to a feature that minimizes the search space for a factor of the polynomial actually guides and produces the learning. Credit is also given to "good" heuristics by ordering them according to their importance (importance with respect to the sense of direction).

Waterman's technique for generating heuristics places more emphasis on credit assignment than the other representation techniques. His work deals more directly with machine learning of heuristics and determining which heuristic is responsible for a "bad" play in poker. We hasten to point out here another common characteristic not discussed in the previous section -- the heuristics in each of these three learning systems are placed into some order by the learning mechanism; thus, providing another reason for having heuristics that can be referenced individually. Credit assignment in Waterman's program can cause a production rule to be modified or cause inclusion of a new rule into the set of ordered heuristics. Section 4.2.3 provides a discussion of how his program assigns credit to heuristics.

Credit assignment in Winston's learning system occurs during model building. Relations can be reinforced by attaching must to a relation. The near misses in the training sequence are used to indicate those relations and properties that must and must not be in any example of the concept. Thus credit is assigned by reinforcing those relations that classify the examples either as positive or negative instances of the concept.

5.0 APPROACHING THE PROBLEMS IN LEARNING SYSTEMS

Since the major thrust of this paper is to encourage learning as a problem solving tool, we need to describe how some of the problems mentioned above can be approached. What the author proposes is the development of a learning system composed of components, in program form, that can be reused over and over in different problem spaces with little or no changes to the components. Some of the problems such as supplying particular information, in the form of rules, etc., for each problem domain still remain since current technology has not been developed to the point that a program can extract this information without help. We are not going to repeat our discussion on the importance of heuristic representation because we feel this has been adequately covered in Section 4.0. That discussion should provide the reader with enough information to select a representation of heuristics. Section 3.0 introduces the reader to problems in learning and how to handle some of them.

Most of the components of the learning system outlined below are in all three learning programs discussed in this paper. We believe a reusable learning system should be composed of the following components or have the following characteristics:

1. A classification mechanism capable of classifying objects into classes so that heuristics appropriate to each class can be applied.

2. Localized learning associated with each class of objects in the classification mechanism so that global learning is not used.
3. A method for representing powerful heuristics that can be created, modified, and executed dynamically.
4. A technique for encoding heuristics to conserve storage.
5. A simple procedure for referencing an individual heuristic (or a set of heuristics) and executing it (or them).
6. At least one type of learning mechanism, e.g. generalized learning and/or concept learning (preferably both types).
7. A procedure for allowing the learning mechanism(s) to direct the creation and modification of heuristics.
8. A procedure(s) capable of analyzing the results of a problem solution attempt to determine if any heuristics should be modified.

Of the components listed above probably only (8) would need to be modified from one application to the next.

6.0 SUMMARY

This paper outlines some of the problems associated with implementing learning programs. We have stressed the importance of the choice of representation of heuristics for this choice affects the learning capabilities of any learning system. Three representations for heuristics were discussed in detail. A brief comparison of these techniques show that they have most of the important requirements of heuristics in common. Of considerable interest is the departure from the simple linear evaluation function approach in the early 1960's to the more powerful languages approach in the early 1970's.

Section 5.0 provides an outline of a learning system of reusable components. Learning programs, in general, are large and time consuming to develop. Thus, a possible approach to using learning is to reuse components without modifying

them when moving to different application areas. Of course some programming effort, dependent on the application, is still required, but the main components of the learning system remain unchanged.

REFERENCES

BEST COPY AVAILABLE

1. Barrow, H. G., Ambler, A. P., and Burstall, R. M. "Some Techniques for Recognising Structures in Pictures", Frontiers of Pattern Recognition, Watanabe, Satosi, (ed.), Academic Press, 1972, pp. 1-29.

2. Claybrook, B. G. POLYFACT: A Learning Program that Factors Multivariable Polynomials, Dissertation, Computer Science/Operations Research Center, Southern Methodist University, 1972, 194 pp.

3. Claybrook, B. G. and Nance, R. E. "The Dynamic Creation and Modification of Heuristics in a Learning Program", In Preparation.

4. Feigenbaum, Edward, and Feldman, J. (eds). Computers and Thought, McGraw-Hill Book Company, 1963, 535 pp.

5. Hormann, A. M. "GAKU: An Artificial Student", Behavorial Science, Vol. 10, pp. 88-107.

6. Hunt, Earl B., Marin, Janet, and Stone, Philip J. Experiments in Induction, Academic Press, New York, 1966, 247 pp.

7. Mendelson, Elliott. Introduction to Mathematical Logic, Van Nostrand Reinhold, New York, 1964, 300 pp.

8. Michie, D. "Strategy - Building with the Graph Traverser", Machine Intelligence 1, 1967, pp. 135-152.

9. Michie, Donald and Ross, Robert. "Experiments with the Adaptive Graph Traverser", Machine Intelligence 5, Meltzer, Bernard and Michie, Donald (eds), American Elsevier, 1970, pp. 301-320.

10. Minsky, M. L. "Steps Toward Artificial Intelligence", Proceedings of IRE 49, 1961, pp. 8-30.

11. Newell, Allen, Shaw, J. C. and Simon, H. A. "A Variety of Intelligent Learning in a General Problem Solver", Self-Organizing Systems, Yovits, Marshall and Cameron, Scott (eds.), Pergamon Press, 1960, pp. 153-189.

12. Newell, Allen and Simon, H. A. "GPS, A Program That Simulates Human Thought", Computers and Thought, Feigenbaum, E. and Feldman, J. (eds.) McGraw-Hill, 1963, pp. 279-293.

13. Nilsson, Nils J. Learning Machines, McGraw-Hill, 1965, 132 pp.

14. Nilsson, Nils J. Problem-Solving Methods in Artificial Intelligence, McGraw-Hill Book Company, 1971, 255 pp.

15. Polya, G. Induction and Analogy in Mathematics, Princeton University Press, Princeton, 1954, 180 pp.

16. Sammet, J. E. "Challenge to Artificial Intelligence: Programming Problems to be Solved", Proceedings of Second International Joint Conference on Artificial Intelligence, September 1971, pp. 59-65.

17. Samuel, A. L. "Some Studies in Machine Learning Using the Game of Checkers", Computers and Thought, Feigenbaum, E. and Feldman, J. (eds.), McGraw-Hill, 1963, pp. 71-105.

18. Slagle, J. R. and Farrell, C. D. "Experiments in Automatic Learning for a Multipurpose Heuristic Program", CACM, Vol. 14, No. 2, pp. 91-99.

19. Slagle, J. R. Artificial Intelligence: The Heuristic Programming Approach, McGraw-Hill Book Company, 1971, 196 pp.

20. Towster, Edwin. "Several Methods of Concept-Formation by Computer", Dissertation, University of Wisconsin, 1970.

21. Waterman, D. A. "Generalization Learning Techniques for Automating the Learning of Heuristics", Artificial Intelligence 1, 1970, pp. 121-170.

22. Winston, P. "Learning Structural Descriptions from Examples", A. I. Technical Report 231, Artificial Intelligence Laboratory, Cambridge, Massachusetts, M. I. T.

23. Winston, P. "The MIT Robot", Machine Intelligence 7, Meltzer, B. and Michie, D. (eds.), Edinburgh University Press, 1972, pp. 431-463.